



La Sécurité

Lotfi Khediri



Sécurité d'une application symfony

- Le contrôle de la sécurité sous Symfony est très avancé mais également très simple. Pour cela Symfony distingue :
 - **l'authentification**: c'est le procédé qui permet de déterminer qui est votre visiteur. c'est le **firewall** qui prend en charge l'authentification. Il y a deux cas possibles :
 - le visiteur est anonyme car il ne s'est pas identifié,
 - le visiteur est membre de votre site car il s'est identifié.
 - **l'autorisation** : intervient après l'authentification, c'est la procédure qui va **accorder les droits d'accès** à un contenu. Sous Symfony, c'est l'**access control** qui prend en charge l'autorisation.



Installation

- Exécutez cette commande pour installer la fonction de sécurité avant de l'utiliser:

\$composer require symfony/security-bundle

A decorative graphic on the left side of the slide. It features a solid red arrow pointing to the right, positioned horizontally. Behind the arrow and extending upwards and to the right are several thin, dark, curved lines that sweep across the frame.

Authentication

Etape 1 : Création de la classe d'utilisateurs

```
$ php bin/console make:user
```

```
The name of the security user class (e.g. User) [User]:
```

```
> User
```

```
Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
```

```
> yes
```

```
Enter a property name that will be the unique "display" name for the user (e.g.  
email, username, uuid) [email]
```

```
> email
```

```
Does this app need to hash/check user passwords? (yes/no) [yes]:
```

```
> yes
```

Etape 1 : Création de la classe d'utilisateurs

- Cette étape est indépendante de la :
 - Méthode d'authentification (par exemple, formulaire de connexion ou jetons d'API) et
 - Façon avec laquelle les utilisateurs sont stockés (base de données, authentification unique)
- Créez une **classe "User"** en utilisant la commande **make**
\$php bin/console make:user
- Une seule règle : elle doit implémentée UserInterface
- On peut ajouter *tout* autre champ ou logique dont on a besoin (utiliser la commande make:entity) à la classe **"User"**
- Effectuer et exécuter une **migration** pour la nouvelle entité.



Etape 2 : Le "User Provider"

- Besoin d'un "fournisseur d'utilisateur": une classe qui permet d'effectuer quelques tâches comme le rechargement des données utilisateur à partir de la session et certaines fonctionnalités optionnelles telque "remember me"
- la commande **make:user** en a déjà configuré un pour nous dans le fichier **security.yaml** sous la clé **providers**
- Si la classe `User` est une entité, rien d'autre à faire.
- Sinon **make:user** générera également une classe **UserProvider** qu'on doit terminer.

En savoir plus sur les fournisseurs d'utilisateurs ici: [Fournisseurs d'utilisateurs](#).

Encodage des mots de passe

- Dans **security.yaml**. On peut contrôler la façon dont les mots de passe sont encodés.
- La commande **make:user** a déjà pré-configuré ceci pour nous.

```
# config/packages/security.yaml
security:
    # ...

    encoders:
        # use your user class name here
        App\Entity\User:
            # Use native password encoder
            # This value auto-selects the best possible hashing algorithm
            # (i.e. Sodium when available).
            algorithm: auto
```


Encodage des mots de passe

- Utiliser le service **UserPassword EncoderInterface** pour encoder les mots de passes avant d'enregistrer les utilisateurs dans la base de données
- Possibilité d'encoder manuellement un mot de passe en exécutant:

```
$ php bin/console security:encode-password
```

```
class UserFixtures extends Fixture
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }

    public function load(ObjectManager $manager)
    {
        $user = new User();
        // ...

        $user->setPassword($this->passwordEncoder->encodePassword(
            $user,
            'the_new_password'
        ));

        // ...
    }
}
```

Authentication et pare-feu

- Le système de sécurité est configuré dans `config/packages/security.yaml`.
- La section la *plus* importante est **firewalls**.

```
security:
  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt)|css|images|js)/
      security: false
    main:
      anonymous: lazy
```





Authentification et pare-feu

- Un "pare-feu" (firewall) est le système **d'authentification**.
- Sa configuration définit **comment** les utilisateurs sont authentifiés (Par exemple, formulaire de connexion, jeton d'API, etc.)
- Un seul pare-feu **est actif** sur chaque **demande**: Symfony utilise la clé **pattern** pour trouver la première correspondance.
- Toutes *les* URL *réelles* sont gérées par le pare-feu **main**
- Aucune clé pattern correspond à *toutes les* URL
- Un pare-feu peut avoir de nombreux modes d'authentification

Authentication : Le mode **anonyme**

- Le mode **anonymos**, s'il est **activé**, permet aux demandes *anonymes* de passer à travers le pare-feu afin que les utilisateurs puissent accéder aux pages publiques, sans avoir besoin de se connecter

				Logged in as	anon.		
				Authenticated	Yes		
				Token class	AnonymousToken		
				Firewall name	default		
200	@ homepage	316 ms	2.0 MB	 anon.	 3 ms		



Authentification des utilisateurs

- Pour gérer la connexion, il faut activer un **Authentificateur**
- un **Authentificateur** = du code qui s'exécute automatiquement avant l'appel de votre contrôleur.
- Créer un **Guard Authenticator** : une classe qui vous permet de **contrôler** chaque partie du processus d'authentification.
- Symfony offre une commande qui simplifie la création d'un Guard Authenticator par **un formulaire de login**
bin/console make:auth

Création d'un "Form Login Authenticator"

```
$ php bin/console make:auth
```

```
What style of authentication do you want? [Empty authenticator]:
```

```
[0] Empty authenticator
```

```
[1] Login form authenticator
```

```
> 1
```

```
The class name of the authenticator to create (e.g. AppCustomAuthenticator):
```

```
> LoginFormAuthenticator
```

```
Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
```

```
> SecurityController
```

```
Do you want to generate a '/logout' URL? (yes/no) [yes]:
```

```
> yes
```




Création d'un "Form Login Authenticator"

La commande précédente génère les éléments suivants:

1. Une route et un contrôleur de connexion,
2. Un template qui rend le formulaire de connexion,
3. une classe d' authenticateur Guard qui traite la soumission de connexion
4. Et met à jour le fichier de configuration de sécurité principal.

Étape 1 : La route "/login" et contrôleur

```
class SecurityController extends AbstractController
{
  / # config/packages/security.yaml
  security:
    # ...

    access_control:
      - { path: ^/login$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
      # ...

  return $this->render('security/login.html.twig', [
    'last_username' => $lastUsername,
    'error' => $error
  ]);
}
```

Étape 2 : Le Template

- Génération d'un simple formulaire HTML traditionnel qui se soumet à **/login**:

```
{% block title %}Log in!{% endblock %}

{% block body %}
<form method="post">
  {% if error %}
    <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security')}}
  {% endif %}

  <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
  <label for="inputEmail" class="sr-only">Email</label>
  <input type="email" value="{{ last_username }}" name="email" id="inputEmail" class="form-control">
  <label for="inputPassword" class="sr-only">Password</label>
  <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password">
</form>
{% endblock %}
```

Étape 3 : L'authentificateur

- Génération du **Guard Authenticator** qui traite le formulaire de login:
- Classe qui contient les méthodes suivantes :
 - public function **supports** (Request \$request)
 - public function **getCredentials** (Request \$request)
 - public function **getUser** (\$credentials, UserProviderInterface \$userProvider)
 - public function **checkCredentials** (\$credentials, UserInterface \$user)
 - public function **onAuthenticationSuccess** (Request \$request, TokenInterface \$token, \$providerKey)
 - protected function **getLoginUrl** ()



Guard Authenticators

- Un authentificateur Guard est une **classe** qui vous donne ***un contrôle complet*** sur votre **processus d'authentification**.
- **Chaque** méthode de cette classe ne contrôle ***qu'une*** petite partie du **processus** d'authentification.
- Il existe de nombreuses façons différentes de créer un authentificateur; voici quelques cas d'utilisation courants:
 - Comment créer un formulaire de connexion
 - Système d'authentification personnalisé avec garde (exemple de **jeton API**)

Étape 5 : Redirection en cas de succès

- Besoin de rediriger l'utilisateur après le succès:

```
// src/Security/LoginFormAuthenticator.php

// ...
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $provider
{
    // ...

-     throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
+     // redirect to some "app_homepage" route - of wherever you want
+     return new RedirectResponse($this->urlGenerator->generate('app_homepage'));
}
```



Processus d'authentification

Les étapes du processus d'authentification

Etape1 : Appel de la méthode **supports()**

- C'est la première méthode appelée à chaque **requête http**.
- Elle retourne :
 - **true** si cette **requête http** contient des informations d'authentification que cet authentificateur sait traiter. Symfony appellera immédiatement **getCredentials()**:
 - **false** Sinon. rien d'autre ne se produit. Symfony n'appelle **aucune** autre méthode de l'authentificateur, et la demande continue comme d'habitude sur notre contrôleur, comme si rien ne s'était passé. Ce n'est pas un échec d'authentification - c'est juste que rien ne se passe du tout.

```
public function supports(Request $request)
{
    return 'app_login' === $request->attributes->get('_route')
        && $request->isMethod('POST');
}
```

Etape 2 : Appel de la méthode **getCredentials()**

- `getCredentials()` : **lit** les informations d'authentification de la demande et les renvoie. Dans ce cas, elle retourne le **email** et **password**. Mais, s'il s'agissait d'un authentificateur avec jeton API, elle retournerait ce jeton.
- Elle renvoie un **tableau** avec une clé **email** définie par `$request->request->get('email')` et **password** définie par `$request->request->get('password')`:
- Symfony appelle **immédiatement** `getUser()` et lui passe ce tableau comme premier argument `$credentials`:

```
public function getCredentials(Request $request)
{
    $credentials = [
        'email' => $request->request->get('email'),
        'password' => $request->request->get('password'),
        'csrf_token' => $request->request->get('_csrf_token'),
    ];
    $request->getSession()->set(
        Security::LAST_USERNAME,
        $credentials['email']
    );

    return $credentials;
}
```

Etape 3 : Appel de la méthode **getUser()**

- Le rôle de `getUser()` consiste à utiliser `$credentials` pour retourner un **objet User**, ou **null** si l'utilisateur n'existe pas.
- Si les utilisateurs sont stockés dans la base de données, nous devons **interroger User** via son e-mail en utilisant `UserRepository` injecté dans le constructeur.
- Si elle renvoie **null**, tout le processus d'authentification s'arrêtera et l'utilisateur verra une **erreur**.
- Mais si elle renvoie un objet **User**, Symfony appelle immédiatement **checkCredentials()**, et il passe le même tableau `$credentials` et l'objet `User`.

```
public function getUser($credentials, UserProviderInterface $userProvider)
{
    ... $token = new CsrfToken('authenticate', $credentials['csrf_token']);
    ... if (!$this->csrfTokenManager->isTokenValid($token)) {
    ...     ... throw new InvalidCsrfTokenException();
    ... }

    ... $user = $this->entityManager->getRepository(User::class)->findOneBy(['email' => $credentials['email']]);

    ... if (!$user) {
    ...     ... // fail authentication with a custom error
    ...     ... throw new CustomUserMessageAuthenticationException('Email could not be found.');
```

Etape 4 : Appel de la méthode **checkCredentials**

- Elle s'occupe de **vérifier si le mot de passe de l'utilisateur est correct** ou tout autre dernier contrôle de sécurité. Elle retourne :
 - **false**, l'authentification échouera et l'utilisateur verrait un message « Informations d'identification non valide ».
 - **true** l'authentification à réussie et maintenant que **l'utilisateur est authentifié, Symfony appelle onAuthenticationSuccess()**:

```
public function checkCredentials($credentials, UserInterface $user)
{
    return $this->passwordEncoder->isPasswordValid($user, $credentials['password']);
}
```


Etape 5 : Appel de la méthode **onAuthenticationSuccess**

- À ce stade, on a **entièrement** rempli **toute** la logique d'authentification.
- En résumé on a utilisé supports() pour dire à Symfony si notre authenticateur doit être utilisé dans cette demande ensuite les informations d'identification sont extraites à partir de la demande, ceux derniers sont utilisés pour retrouver l'utilisateur, et enfin en cas de mot de passe correcte checkCredentials() retourne true.
- Dans ce cas notre utilisateur **est connecté**. Pour un système de connexion par formulaire, on **redirige l'utilisateur vers une autre page**. Pour un système de jetons rien! Laissez simplement la demande continuer comme d'habitude.

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
        return new RedirectResponse($targetPath);
    }

    return new RedirectResponse($this->urlGenerator->generate('product.list'));
    //throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
}
```

En cas d'échec de connexion

- Appel de la méthode **getLoginUrl()**
- Tentative de **redirection** vers l'url retournée par getLoginUrl()
- Dans la plupart des cas l'utilisateur doit être **redirigé vers** la page de connexion (route app_login).

```
protected function getLoginUrl()  
{  
    return $this->urlGenerator->generate('app_login');  
}
```

Authentication et session

- Les informations d'authentification de l'utilisateur sont stockées dans la **session**.
- Au début de chaque requête, ces informations sont chargées depuis la session et nous sommes connectés.
- Lorsqu'on actualise la page, l'objet user est chargé à partir de la session. Mais, nous devons nous assurer que l'objet n'est pas obsolète avec la base de données. C'est le rôle
- C'est donc le travail du fournisseur d'utilisateurs. Lorsque nous actualisons, le fournisseur d'utilisateurs prend l'objet User de la session et utilise l'id pour rechercher un **nouvel** objet User. Tout se passe de manière invisible.

Comment les erreurs d'authentification sont stockées

- L'authentificateur stocke l'erreur dans la session puis nous lisons l'erreur de la session dans notre contrôleur et la rendons dans le template
- Dans SecurityController, nous obtenons l'erreur en appelant la méthode **\$authenticationUtils->getLastAuthenticationError()**

```
public function login(AuthenticationUtils $authenticationUtils): void
{
    // ...

    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();
}
```

```
{% if error %}
    <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}
{% endif %}
```



Les Autorisations



Introduction

- Le processus d'autorisation comporte deux volets différents:
 - L'utilisateur reçoit un ensemble spécifique de **rôles** lors de la connexion (par exemple ROLE_ADMIN).
 - Vous ajoutez du code afin qu'une ressource (par exemple URL, contrôleur) nécessite un "attribut" spécifique (le plus souvent un rôle comme ROLE_ADMIN) pour être accessible.



Les Rôles

- Dans la classe User que nous avons générée précédemment, les rôles sont un tableau stocké dans la base de données et chaque utilisateur se voit *toujours* attribuer au moins un rôle ROLE_USER
- Chaque rôle **doit commencer par ROLE_** (sinon, les choses ne fonctionneront pas comme prévu)
- Autre que la règle ci-dessus, un rôle n'est qu'une chaîne et vous pouvez inventer ce dont vous avez besoin (par exemple ROLE_PRODUCT_ADMIN).
- Dans la classe User les rôles sont affectés avec la méthode **setRoles(array \$roles)**
- Ces rôles sont utilisés pour accorder l'accès à des sections spécifiques de votre site.
- On peut également utiliser une hiérarchie de rôles dans laquelle certains rôles vous donnent automatiquement les droits d'autres rôles.



Ajouter du code pour refuser l'accès

2 manières

- **Méthode 1** : Avec les **access_control** dans security.yaml : permet de protéger les modèles d'URL (par exemple /admin/*). Plus simple, mais moins flexible;
- **Méthode 2** : Dans le contrôleur (ou autre code) .

Méthode 2 : **access_control**

```
# config/packages/security.yaml
security:
    # ...

    access_control:
        # matches /admin/users/*
        - { path: '^/admin/users', roles: ROLE_SUPER_ADMIN }

        # matches /admin/* except for anything matching the above rule
        - { path: '^/admin', roles: ROLE_ADMIN }
```

- On peut définir autant de modèles d'URL que qu'on souhaite
- chacun est une expression régulière.
- **MAIS**, **une** seule sera mise en correspondance par demande:
- Symfony démarre en haut de la liste et s'arrête lorsqu'il trouve la première correspondance:

Voir [Comment fonctionne la sécurité access_control?](#)

Méthode 2 : Dans les contrôleurs ou autre code

- ➡ on peut refuser l'accès **depuis l'intérieur** d'un contrôleur:

```
public function adminDashboard()
{
    $this->denyAccessUnlessGranted('ROLE_ADMIN');

    // or add an optional message - seen by developers
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a page without
}
```

Méthode 2 : Dans les contrôleurs ou autre code

```
+ use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

+ /**
+  * Require ROLE_ADMIN for *every* controller method in this class.
+  *
+  * @IsGranted("ROLE_ADMIN")
+  */
class AdminController extends AbstractController
{
+  /**
+   * Require ROLE_ADMIN for only this controller method.
+   *
+   * @IsGranted("ROLE_ADMIN")
+   */
    public function adminDashboard()
    {
        // ...
    }
}
```

- On peut également sécuriser votre contrôleur **à l'aide d'annotations** (Grâce au SensioFrameworkExtra Bundle)

Contrôle d'accès dans les templates

- Si on souhaite vérifier si l'utilisateur actuel a un certain rôle, on peut utiliser la fonction **is_granted()** dans n'importe quel template Twig

```
{% if is_granted('ROLE_ADMIN') %}  
    <a href="...">Delete</a>  
{% endif %}
```




Vérifier si un utilisateur est connecté (IS_AUTHENTICATED_FULLY)

- Sion veut *uniquement* vérifier si un utilisateur est connecté sans se soucier des rôles

```
public function adminDashboard()  
{  
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');  
  
    // ...  
}
```



Récupérer l'objet utilisateur(connecté)

Dans les controleurs

- Après l'authentification, l'objet User de l'utilisateur actuel est accessible via le raccourci getUser():

```
public function index()
{
    // usually you'll want to make sure the user is authenticated first
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

    // returns your User object, or null if the user is not authenticated
    // use inline documentation to tell your editor your exact User class
    /** @var \App\Entity\User $user */
    $user = $this->getUser();
}
```

Dans les services

```
class ExampleService
{
    private $security;

    public function __construct(Security $security)
    {
        // Avoid calling getUser() in the constructor: auth may not
        // be complete yet. Instead, store the entire Security object.
        $this->security = $security;
    }

    public function someMethod()
    {
        // returns User object or null if not authenticated
        $user = $this->security->getUser();
    }
}
```

- Si on a besoin d'obtenir l'utilisateur connecté dans un service, il faut utiliser le service: Security

Dans les templates

- Dans un template Twig, l'objet utilisateur est disponible via la variable **app.user** grâce à la variable d'application globale Twig "**app**" :

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}  
    <p>Email: {{ app.user.email }}</p>  
{% endif %}
```